REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1996	technical

4. TITLE AND SUBTITLE

OZONE Temporal Constraint Propagator

5. FUNDING NUMBERS

6. AUTHOR(S)

Ora Lassila

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

CMU-RI-TR-96-12

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release;
Distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes the temporal constraint propagator of the OZONE framework for planning and scheduling applications. The role of the propagator is to maintain temporal consistency in networks of activities, enforcing temporal constraints and limiting the search needed when generating a schedule. Unlike some other time bound propagators based on various shortest path algorithms, this one is based on the well-known AC-3 arc consistency algorithm by A.K. Mackworth.

In addition to documenting the constraint propagation architecture of OZONE and the functional requirements of the propagator, as well as describing the actual propagation algorithm, this report serves as a programmer's reference to the functional interface to the propagator. It also gives has notes about the internal design of the system and documents the most important internal functions.

14. SUBJECT TERMS

15. NUMBER OF PAGES
30 pp

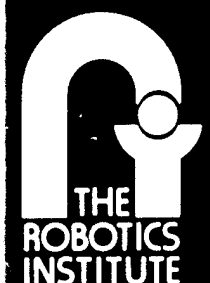
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unlimited	unlimited	unlimited	unlimited

OZONE Temporal Constraint Propagator

Ora Lassila

CMU-RI-TR-96-12



Carnegie Mellon University

The Robotics Institute

Technical Report

OZONE Temporal Constraint Propagator

Ora Lassila

CMU-RI-TR-96-12

The Robotics Institute *
Carnegie Mellon University
Pittsburgh, PA 15213

March 1996

* This research has been supported in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Material Command, USAF, under grant numbers F30602-90-C-0119 and F30602-95-1-0018 as well as F30602-91-C-0014 (under subcontract to BBN Systems and Technologies) as part of the ARPA/Rome Labs Planning Initiative), and the CMU Robotics Institute. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency and Rome Laboratory or the U.S. Government

Abstract

This report describes the temporal constraint propagator of the OZONE framework for planning and scheduling applications. The role of the propagator is to maintain temporal consistency in networks of activities, enforcing temporal constraints and limiting the search needed when generating a schedule. Unlike some other time bound propagators based on various shortest path algorithms, this one is based on the well-known AC-3 arc consistency algorithm by A.K.Mackworth.

In addition to documenting the constraint propagation architecture of OZONE and the functional requirements of the propagator, as well as describing the actual propagation algorithm, this report serves as a programmer's reference to the functional interface to the propagator. It also gives has notes about the internal design of the system and documents the most important internal functions.

Acknowledgements

The development of the temporal constraint propagator described in this report was supported in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Material Command, USAF, under grant numbers F30602-90-C-0119 and F30602-95-1-0018 as well as F30602-91-C-0014 (under subcontract to BBN Systems and Technologies) as part of the ARPA/Rome Labs Planning Initiative, and the CMU Robotics Institute.

The design of the propagator benefited a great deal from my experience (at the Helsinki University of Technology) with the design and implementation of the DOM scheduler, from numerous conversations with my former colleague Seppo Törmä (who was the principal architect of the DOM Time Bound Propagator), as well as from intellectual interaction with Robert Aarts. In addition, I am grateful to Marcel Becker for his persistence in trying to find bugs from the propagator; my apologies that not many were to be found.

I would also like to thank my wife Marcia whose perceptive understanding of production planning issues from the management and business perspectives has always helped me see the "real world" more clearly and not forget why I design software.

Pittsburgh, March 1996

Ora Lassila

Contents

Chapter 1

Introduction

This report will provide a description of the temporal constraint propagator (the "Time Bound Propagator", or TBP for short) of the OZONE[1] Planning and Scheduling Toolkit [8]. The TBP is one of the core services provided by the OZONE toolkit. Its purpose is to maintain temporal consistency in networks of activities, enforcing temporal constraints and limiting the search needed when generating a schedule.

This manual consists of four separate parts: This chapter will give an overview of the constraint propagation architecture of OZONE, as well as outline the design requirements for the TBP. Chapter 2 will describe the actual propagation algorithm used. Chapter 3 documents the API available for calling the TBP, and Chapter 4 documents some of the internal functions of the system.

1.1 Functional Requirements

The current TBP of OZONE represents an extension of capabilities compared to the old OPIS TBP[6]. The functional requirements for the new propagator are:

1. Ability to **represent points in time as intervals**, and constraints – including operation duration constraints – in terms of upper and lower bounds, with possibly an infinite upper bound.
2. Ability to **propagate changes in time bounds in activity graphs**. These structures are directed acyclic graphs with possibly multiple sources and sinks. Cycles are not allowed.

[1] "OZONE" = O_3 = "Object-Oriented OPIS," previously "DITOPS."

3. Ability to **propagate changes in time bounds in a hierarchical representation of activities**. Hierarchies may consist of OR-nodes (alternatives) and AND-nodes (aggregation).

4. Ability to handle **relaxation of time bounds** in case of infeasible constraints.

5. Ability to **initialize the time bounds** of operations and operation graphs after these structures have been created.

Requirement 2 represents an improvement over the old propagator which was only capable of handling linear operation lineups. Requirement 4 is optional and has been implemented in a way which allows the domain modeler to choose whether to use this feature or not.

In comparison to Tachyon [11] – another temporal reasoning tool developed in the ARPA/Rome Labs Planning Initiative – we can make the following observations with respect to the design desiderata for Tachyon as outlined by Stillman (the reader is referred to [11] for details):

- Ability to **deal with uncertainty**: OZONE can handle the same way Tachyon does since all time points are represented by intervals (requirement 1 above).
- Ability to express both **qualitative and quantitative constraints**: again, OZONE handles this in the same way Tachyon does (requirements 1 and 2).
- Ability to express **parameterized qualitative constraints**: ditto (requirement 1).
- Ability to provide **multiple granularities of time**: OZONE's time representation allows this.
- OZONE can also handle **hierarchical** representations of activities which Tachyon cannot (requirement 3).

1.2 Constraint Propagation Architecture

In OZONE, temporal constraint propagation functionality is separate from basic activity representation. The base class for activity modeling, `operation`, makes no assumptions about the TBP; in fact, it does not even make assumptions about start- and end-times in general. The basic idea in OZONE is that when a domain model is built, appropriate *mixin* classes are mixed with the `operation` class, providing the resulting activity class the ability to manage time bounds and temporal constraints. The framework provides a protocol for enforcing temporal consistency (these functions, as well as other functions in the TBP API are described in Chapter 3).

The following is a list of (mostly mixin) classes which can be used when building a domain model and associated temporal constraint management:

- `earliest-latest-mixin` is a mixin class which provides four *time bounds* for activities: the earliest and latest start times as well as the earliest and latest end times (one can think of this as both start- and end-times being represented by an interval).
- `relation-owner-mixin` is mixin class which provides activities with the ability to "own" temporal relations, in other words to be linked to each other using temporal relations (the base class for temporal relations is called `relation`).
- `monotonic-tbp-mixin` is a mixin class which implements the basic OZONE propagation algorithm, a version of Mackworth's AC-3 [7] modified for continuous, monotonic domains (see Chapter 2 for a description of this algorithm). This class actually inherits from both `earliest-latest-mixin` and `relation-owner-mixin`.
- `relation` is the base class for temporal relations. Subclasses of this class include `before`, `same-start` and `same-end`. Normally, these relation classes do not need to be subclassed in a domain model.

1.3 Applications of OZONE TBP

The Time Bound Propagator described in this document has been used in various applications built using the OZONE Planning and Scheduling Framework. As the standard temporal constraint propagator for OZONE, the propagator has been used with the domain models built for the DITOPS scheduler, as well as with some other more diverse planning applications. These include the following:

- The first application of the propagator was with the **IFD3 Plan Feasibility Checker** built as a component of the IFD3 TARGET system (for the ARPA/Rome Labs Planning Initiative), and performed feasibility checks on plans created by TARGET, a decision support tool for authoring military course-of-action plans (see, for example, [2]).
- The **DITOPS Transportation Scheduler** has served as a research prototype and technology demonstrator in the ARPA/Rome Labs Planning Initiative, with various domain models for solving strategic deployment problems encountered by the US Transportation Command (TRANSCOM), including models of the IFD2 scenarios [9].

- The **DITOPS Medical Evacuation Planner** is a prototype planning application for reactive maintenance of aeromedical evacuation plans. In this application the role of the propagator is critical as it is used for detecting inconsistencies in patient itineraries [4].
- A model originally built for the OPIS scheduler of a Westinghouse turbine plant was ported to OZONE, demonstrating – among other things – the suitability of the propagator for maintaining temporal information about manufacturing process routings.

Chapter 2

Propagation Algorithm

The propagation algorithm has two distinct but interacting parts: the *horizontal* propagator and the *vertical* propagator.

- The role of the **horizontal propagator** is to propagate changes in time-bounds in an operation graph. The graph is understood to consist of time points connected by temporal relations. Since OZONE represents activities as single entities, activities are thought of as relations – duration constraints – between their respective start- and end times.
- The **vertical propagator** operates on *hierarchical* representations of activities. OZONE recognizes two different kinds of hierarchical relations between an activity and its "children": The OR relation represents alternatives, the children are understood to be alternative representations of the parent activity; the AND relation represents aggregation, where the children form an operation graph which represents the substructure of the parent operation. The role of the vertical propagator is to propagate changes up and/or down in this hierarchy, calling the horizontal propagator to handle the lateral propagation.

The system can handle a number of different kinds of temporal relations. each temporal relation class can be used to represent a constraint between a start- or end-time of an activity and a start- or end-time of another activity. Temporal relations also have metric bounds, allowing for minimum and maximum separation of the constrained time points. The temporal relations currently handled by the system (and their correspondence to relations defined by Allen [1]) are:

- *Before* is a relation between the end-time of an activity and the start-time of another. With respect to Allen's relations, it can be used to implement both "before" and "meets."

- *Same-start* is a relation between the start-times of two activities. It can be used for Allen's "starts" and "overlaps" -relations.
- *Same-end* is a relation between the end-times of two activities. It can be used to implement "finishes" and "overlaps."

Together the last two can be used to implement "equal" and "during", thus the available relations cover all of Allen's interval relations.

2.1 Horizontal Propagator

The horizontal propagator is basically an arc consistency algorithm for temporal relations, based on Mackworth's AC-3 -algorithm [7], and very much akin to the propagation algorithm of Törmä [12] used in the DOM scheduler [5, 13]. The general idea of the horizontal propagation algorithm is given below. Please note that since activities themselves are thought of as temporal relations between start- and end-times (i.e., duration constraints), the algorithm can operate in terms of time points and arcs between them.

Algorithm TBP: Given an activity graph G consisting of time points and relations $\{r\}$, change the bounds of the time points so that all relations are consistent.

1. Establish an empty queue Q of relations (arcs), and place some relations from the activity graph G into the queue (see an explanation below about which relations to pick).
2. Remove first relation r from the queue Q, and make it consistent (see explanation below about how consistency is achieved).
3. If there was a change in time bounds due to making r consistent, place all affected relations from G into queue Q. Any relations already in the queue are ignored.
4. If the queue Q is not empty, loop back to step 2.

The current implementation of the propagator (class `monotonic-tbp-mixin`) operates on time bounds in a *monotonic* fashion. In other words, during propagation the time bounds can only "move" in one direction. When a relation is made consistent, the following considerations are important: (1) Constraints are arcs between time points, (2) time points (in the current implementation) consist of an upper and a lower bound (i.e., earliest and latest), and (3) for any invocation of the propagator, either earliest or latest bounds are modified.

Figure 2.1 depicts an example of propagation from time bound A through relation r to time bound B. Given a minimum separation of d for r, the relation is made consistent (after a positive change in A) as follows:

- If $A + d \leq B$ then do nothing.
- If the activity that "owns" B has already been scheduled, time bounds cannot be changed: signal a time bound conflict.
- Otherwise, change B to $A + d$ and add all relations involving B to the queue.

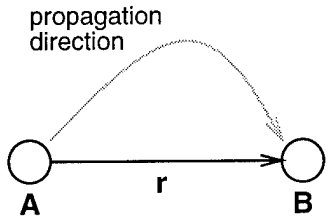


Figure 2.1: Relation Propagation Example

If the original change had taken place in B (positive change, i.e. still going forward), the relation r instead of "pushing" now "pulls" A (if A is further away from B than the *maximum* separation of r). In general, propagation can proceed either forward or backward, and depending on which way relations are traversed, they either push or pull other time bounds. Because OZONE represents activity graphs in terms of activities and not in terms of time points, each relation type has specific "propagation" behavior, effectively implementing the time point -based algorithm.

2.1.1 Relaxation of Time Bounds

The TBP has the capability of relaxing time bounds in a situation where a feasible configuration of bounds cannot be found. Relaxation can only happen "forward", that is, the propagator will relax end-times.

When the propagator is proceeding backwards (i.e., tightening end-times), and cannot (because resource unavailability) establish feasible bounds, relaxation is optional: The propagator is called recursively to propagate forward, disregarding any posted due dates. After the forward propagation the backward propagation is resumed. What happens in actuality is that the propagator is trying to "push" bounds backward, and failing to do so pushes them forward to allow for a feasible solution.

2.2 Vertical Propagator

The vertical propagator is a control structure on top of the horizontal propagator. To understand how it works we first have to review the modeling assumptions. An activity hierarchy in OZONE consists of (currently) three types of activities:

- OR-nodes represent alternatives. For example, Figure 2.2 depicts an activity node N which actually consists of three alternatives N1, N2 and N3. When this activity gets scheduled, one (and only one) of the alternatives from its substructure gets picked (typically these represent resource alternatives, but they could also represent alternative routings if N1, N2 and N3 had any substructure).
- AND-nodes represent entire activity graphs. For example, Figure 2.3 depicts an activity node N which actually represents an activity graph consisting of activities N1, N2 and N3. When this activity gets scheduled, all of the activities in the substructure have to be scheduled. Any number of the nodes N1, N2, etc. may be sources and/or sinks of the directed acyclic graph node N represents (in fact, N may represent a set of acyclic graphs). Activity graphs in OZONE are formed by connecting activities with temporal relations.
- Leaf nodes represent individual activities. They have no substructure.

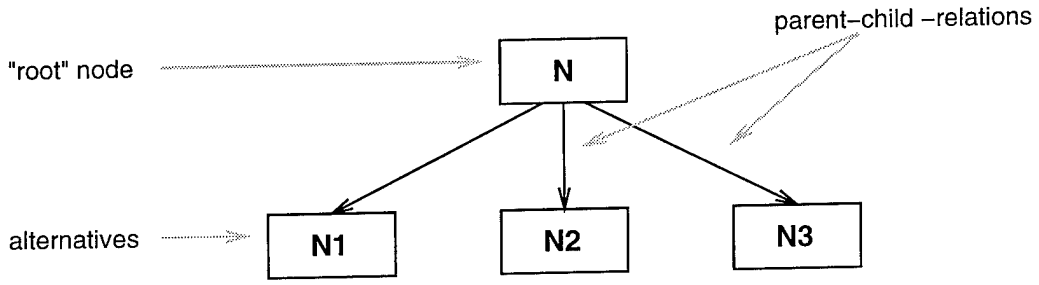


Figure 2.2: Example of an OR-node

Please note that temporal relations are always placed between nodes sharing the same parent (thus they are also at the same level in an activity hierarchy).

The vertical propagator gets invoked in two different situations: When (1) an activity or activity graph has been propagated horizontally, the influences are then propagated *upwards*, and (2) during horizontal propagation when either an AND-node or an OR-node is encountered, their time bounds are verified by propagating at a lower level.

When propagating up, the behavior is different for AND- and OR-nodes:

- For AND-nodes, when propagating forward a candidate start time is simply the minimum of their children's start times; the horizontal propagator is then called to propagate the potential change in start time. When propagating backward a candidate end time is the maximum of children's end times; the horizontal propagator is then called to propagate the potential change in end time.
- For OR-nodes, the new start end end times are the minimum start time and maximum end time of their children (or if the activity from which upward propagation was invoked has already been scheduled, simply the start and end times of this activity). The horizontal propagator is called after all relations linked to the OR-node have been queued.

When propagating downwards, the horizontal propagator is simply called for all *child roots* of a node. For OR-nodes, all children are roots; for AND-nodes, two sets of roots are maintained: *sources* or forward roots, and *sinks* or backward roots.

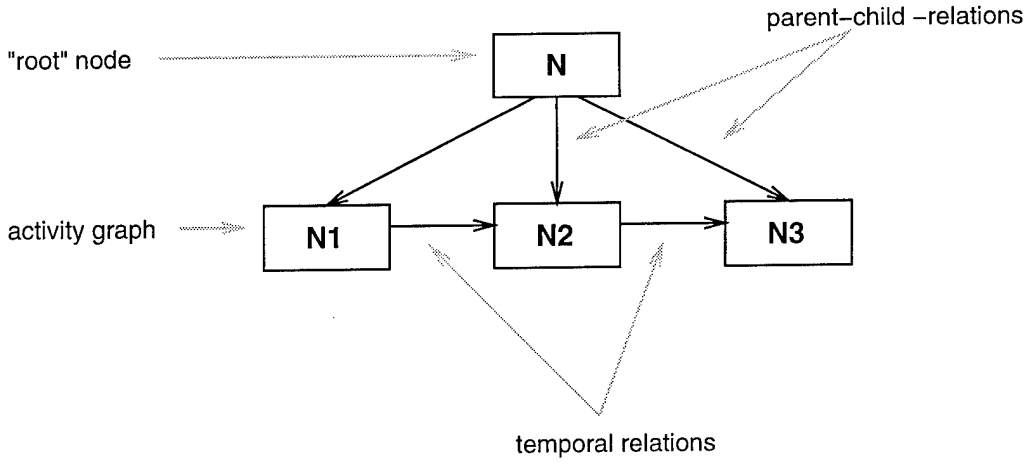


Figure 2.3: Example of an AND-node

Chapter 3

Propagator API

This chapter describes the API for the Time Bound Propagator. The actual propagator has been implemented as an operation mixin class `monotonic-tbp-mixin`. Relations (base class `relation`) each have their own propagation method; for reasons of consistency, operations (that is, instances of `monotonic-tbp-mixin`) are also treated as temporal relations between their starting and ending points.

3.1 Essential Propagator API

This section documents the essential, most frequently called functions and variables of the propagator.

`operation-update-time-bounds` [Generic function]

operation
`&key` *point-mode*
change-mode
upwardsp

This generic function calls the propagator to update the time bounds of *operation* (assuming the state of *operation* has changed) and other associated operations. It uses information about the operations' temporal relations to deduce which operations are affected by the change.

The parameter *point-mode* should be either `:earliest` (the default) or `:latest`, and will determine whether the time bound propagator will propagate earliest or latest times.

The parameter *change-mode* should be either `:increase` (the default) or `:decrease`,

giving the direction of propagation.

The parameter *duration-mode* should be either `:min` (the default) or `:max`, indicating the type of durations that are to be used during propagation.

The function `operation-update-time-bounds` supports three different modes of operation: (1) the default mode (when no keyword parameters are passed) consists of two propagation passes, one with `:earliest` and `:increase`, the other with `:latest` and `:decrease` (`:min` duration mode is used); (2) the initialization mode (when initial start time *st* and end time *ft* are passed) is like the default mode, but bounds are first initialized (end time relaxation may also happen); and (3) the specific mode (when *-mode* parameters are passed) consists of a single pass with specific mode parameters.

`*tbp-check-resources-p*` [Variable]

This variable, which defaults to true, is used in deciding whether the propagator should check resource availability when modifying time bounds (i.e., make calls to `find-available-time`).

`link-operation` [Generic function]
operation
target
relation-class
lower-bound
upper-bound
`&key` *name*

This generic function creates a temporal relation (instantiated from *relation-class*, a subclass of `temporal-relation`) using *lower-bound* and *upper-bound* as the initialization arguments, between *operation* and *target* (both operations). The relation will be added to the relations of *operation* and *target*.

The parameter *name* can be used to name the relation instance (it defaults to `nil`, i.e. no name).

3.2 Propagator Protocol

This section documents the rest of the functional protocol of the Time Bound Propagator. The protocol consists of generic function for which various propagator-related entities have to define methods.

`operation-est` [Generic function]
operation

This function accesses the *earliest start time* of an operation.

`operation-eft` [Generic function]
operation

This function accesses the *earliest finish time* of an operation.

`operation-lst` [Generic function]
operation

This function accesses the *latest start time* of an operation.

`operation-lft` [Generic function]
operation

This function accesses the *latest finish time* of an operation.

`operation-initialize-time-bounds` [Generic function]
operation
`&key` *st*
ft
est
eft
lst
lft
status
`&allow-other-keys`

This function can be used to initialize the time bounds of an operation. Calling this function will only modify *operation* but will not invoke the propagator on any temporal relations connected to *operation*. The keyword parameters *st* and *ft* allow start- and end-times to be initialized without concern to any earliest/latest issues. The parameter *status* allows the operation's scheduled/unscheduled status to be initialized. See the OZONE library manual description for the class `operation` for details.

`tbp-reversible-p` [Generic function]
operation

This function should return true if time bound relaxation is allowed by switching propagation direction when propagating backward.

`operation-relations` [Generic function]
operation

This function accesses the set of temporal relations connected to *operation*.

`operation-forward-child-roots` [Generic function]
operation

This function accesses the set of forward propagation roots of *operation*. These roots are those children of the operation which act as propagation sources when the vertical propagator is invoked.

`operation-backward-child-roots` [Generic function]
operation

This function accesses the set of backward propagation roots of *operation*. These roots are those children of the operation which act as propagation sinks when the vertical propagator is invoked.

`relation-from-operation` [Generic function]
relation

This function accesses the starting point of a relation.

`relation-to-operation` [Generic function]
relation

This function accesses the end point of a relation.

`relation-lower-bound` [Generic function]
relation

This function accesses the minimum separation value of a relation.

`relation-upper-bound` [Generic function]
relation

This function accesses the maximum separation value of a relation (`nil` if no upper bound has been specified).

`queue-relation` [Generic function]
relation
source-to-sink-p

This function queues a relation for subsequent consideration by the propagator. The parameter *source-to-sink-p* indicates the direction of propagation: true indicates start-to-end -direction, `nil` indicates end-to-start -direction.

`relation-affected-by-change-p` [Generic function]
relation
source-to-sink-p
st-changed-p

This is a predicate function for determining whether a particular relation is affected by a given change in time bounds. The *source-to-sink-p* parameter indicates the direction of the propagation, the *st-changed-p* is true is a start time changed, `nil` is an end time

changed. Relation classes should define methods for this generic function to indicate how they relate to the time points of an operation.

`propagate` [Generic function]
entity
source-to-sink-p

Methods of this generic function implement the time bound propagation over temporal intervals.

3.3 Propagator Classes

This section documents the various classes that make up the Time Bound Propagator functionality.

`earliest-latest-mixin` [Class]

Subclasses of this class include `monotonic-tbp-mixin`. This is a mixin class which provides four *time bounds* for activities: the earliest and latest start times as well as the earliest and latest end times (one can think of this as both start- and end-times being represented by an interval).

`operation-est` [Method]
(*self* `earliest-latest-mixin`)
`(setf operation-est)` [Method]
value
(*self* `earliest-latest-mixin`)

These methods access the slot `est`. This slot holds the earliest start time.

`operation-eft` [Method]
(*self* `earliest-latest-mixin`)
`(setf operation-eft)` [Method]
value
(*self* `earliest-latest-mixin`)

These methods access the slot `eft`. This slot holds the earliest finish time.

`operation-lst` [Method]
(*self* `earliest-latest-mixin`)
`(setf operation-lst)` [Method]
value
(*self* `earliest-latest-mixin`)

These methods access the slot `lst`. This slot holds the latest start time.

operation-lft [Method]
    (*self* earliest-latest-mixin)

(setf operation-lft) [Method]
    value
    (*self* earliest-latest-mixin)

These methods access the slot `lft`. This slot holds the latest finish time.

initialize-instance [after method]
    (*self* earliest-latest-mixin)
    &rest *initargs*
    &key *st*
         ft
         est
         eft
         lst
         lft
         status

This method calls `operation-initialize-time-bounds`.

operation-initialize-time-bounds [Method]
    (*self* earliest-latest-mixin)
    &key *st*
         ft
         est
         eft
         lst
         lft
         status
    &allow-other-keys

This method implements the specified functionality of its generic function.

tbp-reversible-p [Method]
    (*operation* earliest-latest-mixin)

This method always returns `nil`.

relation-owner-mixin [Class]

Subclasses of this class include `monotonic-tbp-mixin`. This is mixin class which provides activities with the ability to "own" temporal relations, in other words to be linked to each other using temporal relations (the base class for temporal relations is called `relation`).

Mixing `relation-owner-mixin` with an operation class provides automatic main-

tenance of the forward- and backward-roots required for lower-level propagation. This behavior is implemented using mechanisms from OZONE's `aggregate` class.

`operation-relations` [Method]
    (*self* `relation-owner-mixin`)
`operation-add-relation` [Method]
    value
    (*self* `relation-owner-mixin`)
    `&optional` *updatep*
`operation-remove-relation` [Method]
    value
    (*self* `relation-owner-mixin`)
    `&optional` *updatep*

These methods access the slot `relations`. This slot holds the relations "owned" by this entity.

`operation-forward-child-roots` [Method]
    (*self* `relation-owner-mixin`)
`(setf operation-forward-child-roots)` [Method]
    value
    (*self* `relation-owner-mixin`)

These methods access the slot `forward-child-roots`. This slot holds the "sources" for lower-level propagation.

`operation-backward-child-roots` [Method]
    (*self* `relation-owner-mixin`)
`(setf operation-backward-child-roots)` [Method]
    value
    (*self* `relation-owner-mixin`)

These methods access the slot `backward-child-roots`. This slot holds the "sinks" for lower-level propagation.

`link-operation` [Method]
    (*operation* `relation-owner-mixin`)
    (*target* `relation-owner-mixin`)
    relation-class
    lower-bound
    upper-bound
    `&key` *name*

This method implements the specified functionality of its generic function.

`monotonic-tbp-mixin` [Class]

This class inherits directly from `earliest-latest-mixin` and `relation-owner-mixin`. This mixin class implements the *Time Bound Propagator*. The propagator utilizes *temporal relations* to maintain time windows of feasible allocation intervals. The propagator also utilizies the four slots `est`, `lst`, `eft` and `lft` (from `operation`) that contain the *time bounds* of the operation: the *earliest start time*, the *latest start time*, the *earliest finish time* and the *latest finish time*. After an operation has been scheduled, the `est` and `lft` slots contain the *actual* start and end time of the operation.

Specializing this class provides a way of extending the time bound propagation behavior of the system.

`operation-update-time-bounds` [Method]
(*operation* `monotonic-tbp-mixin`)
`&key` *point-mode*
change-mode
upwardsp
`&allow-other-keys`

This method implements the specified functionality of its generic function.

`tbp-reversible-p` [Method]
(*operation* `monotonic-tbp-mixin`)

This method always returns true.

`queue-relation` [Method]
(*relation* `monotonic-tbp-mixin`)
source-to-sink-p

This method implements the specified functionality of its generic function.

`propagate` [Method]
(*relation* `monotonic-tbp-mixin`)
source-to-sink-p

This method propagates influences from start time to end time (when *source-to-sink-p* is true) or vice versa.

3.4 Temporal Relation Classes

`relation` [Class]

`:from-operation` [Initarg]
`:to-operation` [Initarg]
`:lower-bound` [Initarg]
`:upper-bound` [Initarg]

Subclasses of this class include `same-end`, `same-start` and `before`. This is the base class for all temporal relations. Instances of this class have two slots that represent the lower and upper bounds of the time difference between the start and/or end times of the two operations connected by this relation.

`relation-from-operation` [Method]
`(`*self* `relation)`
`(setf relation-from-operation)` [Method]
value
`(`*self* `relation)`

These methods access the slot `from-operation`. This slot holds the operation from which this relation originates.

`relation-to-operation` [Method]
`(`*self* `relation)`
`(setf relation-to-operation)` [Method]
value
`(`*self* `relation)`

These methods access the slot `to-operation`. This slot holds the operation to which this relation points.

`relation-lower-bound` [Method]
`(`*self* `relation)`
`(setf relation-lower-bound)` [Method]
value
`(`*self* `relation)`

These methods access the slot `lower-bound`. This slot holds the lower bound time difference of the temporal relation. It defaults to `0`.

`relation-upper-bound` [Method]
`(`*self* `relation)`
`(setf relation-upper-bound)` [Method]
value
`(`*self* `relation)`

These methods access the slot `upper-bound`. This slot holds the upper bound time difference of the temporal relation. It defaults to `-time-infinite-`.

`before` [Class]

This class inherits directly from `relation`. Relations of this class represent the difference between the end time of a preceding operation and the start time of a succeeding operation.

`relation-affected-by-change-p` [Method]
(*relation* `before`)
source-to-sink-p
st-changed-p

This method returns *st-changed-p* XOR *source-to-sink-p*.

`propagate` [Method]
(*relation* `before`)
source-to-sink-p

This method implements the specified functionality of its generic function.

`same-start` [Class]

This class inherits directly from `relation`. Relations of this class represent the difference between the start times of two operations.

`relation-affected-by-change-p` [Method]
(*relation* `same-start`)
source-to-sink-p
st-changed-p

This method returns *st-changed-p*.

`propagate` [Method]
(*relation* `same-start`)
source-to-sink-p

This method implements the specified functionality of its generic function.

`same-end` [Class]

This class inherits directly from `relation`. Relations of this class represent the difference between the end times of two operations.

`relation-affected-by-change-p` [Method]
(*relation* `same-end`)
source-to-sink-p
st-changed-p

This method returns the negation of *st-changed-p*.

`propagate` [Method]
(*relation* `same-end`)
source-to-sink-p

This method implements the specified functionality of its generic function.

3.5 Propagator Conflicts

This section describes the various *conflicts* the Time Bound Propagator can signal. In the current implementation conflicts are Common Lisp *conditions*, which allows them to be signalled independent of the control structure or calling context of the library.

Before propagating, an activity graph is analyzed for its topological properties (in actuality the cycle checking takes place during the maintenance of the "child roots"). This may give rise to various conflicts:

(1) "cycle conflicts" indicate that the activity graph is not acyclic. The cycle detection is a simplified topological sort, where all relations are treated equal, activities are assumed to be points in time (that is, to have zero duration) and metric bounds are not taken into account. Sorting works so that we find one activity with no incoming relations, delete that and its outgoing relations, and loop. If no activity can be found with no incoming relations, a cycle is found. In this case, all remaining activities and relations are placed in the cycle conflict object (they contain the cycle, though the set is not minimal).

(2) "disconnected subgraph conflicts" indicate that the graph in question actually consists of several independent subgraphs. This situation is legal and propagator can handle it, but a conflict is signalled just in case.

Conflicts detected during propagation are "time bound violations" (where an earliest start or finish time becomes greater that the corresponding latest time) and "resource unavailability conflicts" (where the required resources for an activity are not available during the inferred time window). Resource unavailability will be detected first, when propagating over a relation. A new start or end time is computed as follows: the relation "pushes" or "pulls" the time point, after which a suitable point is determined by resource availability; if resource is not available within the window, a resource conflict is posted. After successfully computing the start or end time, it is validated by checking it against the corresponding latest time (given we are computing the earliest time, or vice versa). If the earliest advances beyond the latest, a time bound violation is posted.

`tbp-conflict` [Class]

This class inherits directly from `time-conflict` and `common-condition-base`.

Subclasses of this class include `resource-unavailability` and `time-bound-violation`. This is the base class of all conflicts created by the Time Bound Propagator. It exists for ontological purposes, and no specific functionality has been defined for this class.

`time-bound-violation` [Class]

This class inherits directly from `tbp-conflict`. This conflict will be posted when an earliest time bound surpasses a latest time bound.

`resource-unavailability` [Class]
`:st-related-p` [Initarg]

This class inherits directly from `tbp-conflict`. This conflict is posted when (if) the propagator queries for resource availability while validating time bounds and finds the resource in question to be unavailable (i.e., no available capacity on the resource).

`conflict-st-related-p` [Method]
(*self* `resource-unavailability`)

This method accesses the slot `st-related-p`.

`conflict-earliest-bound-p` [Method]
(*self* `resource-unavailability`)

This method accesses the slot `earliest-bound-p`.

`conflict-bound-increase-p` [Method]
(*self* `resource-unavailability`)

This method accesses the slot `bound-increase-p`.

`disconnected-subgraphs` [Class]
`:subgraphs` [Initarg]

This class inherits directly from `solution-structure-conflict`. This conflict is signalled when disconnected subgraphs are detected during relation maintenance. It is possible to ignore this conflict.

`conflict-subgraphs` [Method]
(*self* `disconnected-subgraphs`)

This method accesses the slot `subgraphs`. This slot holds lists which represent sets of activities of the individual subgraphs detected.

`cyclic-operation-graph` [Class]

This class inherits directly from `solution-structure-conflict`. This conflict is signalled when an activity graph is found not to be acyclic. The condition object contains a set of activities which are participating in the detected cycle.

Chapter 4

Internal Implementation

This chapter describes the internal implementation of the monotonic propagation mechanism. These descriptions are relevant when extending this mechanism (i.e., when subclassing `monotonic-tbp-mixin`). Some of these functions may also be used when writing a completely new propagator.

4.1 Propagator Globals

These global variables have a meaningful value only during an invocation of the propagator. It is an error to access them at any other time.

`*tbp-mode-earliest-p*` [Variable]

When true, signifies that the propagator is manipulating earliest bounds (during an invocation either earliest or latest bounds are modified, not both).

`*tbp-mode-increase-p*` [Variable]

When true, signifies that the propagator is modifying time bounds by increasing them (the propagator has monotonic time bound behavior).

`*tbp-max-peeled-capacity*` [Variable]

This variable is used when computing the maximum amount of capacity "peeled" during an invocation of the propagator. It is only used if the propagator has been instructed to check resource availability (i.e., if `*tbp-check-resources-p*` is true).

4.2 Propagation Engine

The "propagation engine" uses a queue of temporal relations which are yet to be "made consistent", i.e. are potentially inconsistent. Any change in a time bound will cause those relations which are affected by the change to be placed in the queue. Any relation already in the queue will not be queued twice.

`tbp` [Macro]
queueing-form

This macro will invoke the propagation engine. The parameter *queueing-form* is a form the execution of which will have the side effect of placing some relations in the propagator queue (either by calling `queue-relation` or `queue-relations`). This macro will establish an empty dynamic binding for the queue, allowing recursive, re-entrant invocations of the propagator.

`queue-relations` [Function]
relations
operation
st-changed-p

This function will queue all relations in *relations*. It is called as a result of change in *operation*, an operation instance. The boolean parameter *st-changed-p* will indicate whether the change affected the start time (true value) or the end-time (`nil` value). Queuing of individual relations is done by calling `queue-relation`.

`*tbp-queue*` [Variable]

This variable holds the propagator queue (a list of queued relations).

`make-qr` [Macro]
relation
source-to-sink-p

This macro – which is called like a function – constructs a queued relation element. In the current implementation this is a cons whose car is the relation and cdr is the *source-to-sink-p* value (indicating the propagation direction).

`qr-relation` [Macro]
qr

This macro – which is called like a function – accesses the relation part of a queued relation.

`qr-source-to-sink-p` [Macro]
qr

This macro – which is called like a function – accesses the propagation direction part of a queued relation.

`qr-eq` [Function]
r1
r2

This predicate is true is two queued relations can be considered the same relation (i.e., both the relation and the propagation direction are the same).

4.3 Time Bound Manipulation

There are several "helper functions" which can be used when manipulating time bounds during propagation.

`operation-st` [Function]
operation

Based on the value of `*tbp-mode-earliest-p*` this function will return the result of either `operation-est` or `operation-lst`.

`(setf operation-st)` [Function]
st
operation

Based on the value of `*tbp-mode-earliest-p*` this function will set the value of either `operation-est` or `operation-lst` to *st*.

`operation-ft` [Function]
operation

Based on the value of `*tbp-mode-earliest-p*` this function will return the result of either `operation-eft` or `operation-lft`.

`(setf operation-ft)` [Function]
ft
operation

Based on the value of `*tbp-mode-earliest-p*` this function will set the value of either `operation-eft` or `operation-lft` to *ft*.

`>tbp` [Function]
x
y

If `*tbp-mode-increase-p*` is true (i.e., the propagation is proceeding forward), this

function is equivalent to the function >. When propagating backward, it is equivalent to the function <.

`propagate-change-p` [Function]
relation
new-time
old-time

The values used internally for *propagation reasons* are either activities (i.e., duration constraints), temporal relations (i.e., precedence constraints) and the value `t`, signifying forced propagation (see the next section on "Propagation Methods." Given a propagation reason *relation*, this predicate function returns true if the time value *new-value*, when replacing *old-value*, should cause changes and subsequent queuing of associated relations. In practice, this function is true if either of the following conditions is true: (1) `>tbp`, when called for *new-value* and *old-value*, returns true, or (2) when *relation* is `t`.

4.4 Propagation Methods

This section documents the generic function and associated methods which implement the propagator.

`propagate-st` [Generic function]
operation
relation
st

Given an activity *operation*, a relation (propagation reason) *relation*, and a new (potential) start time *st*, this function will attempt to change the start time of *operation* and queue any affected relations if change occurs.

`propagate-st` [Method]
(*operation* `monotonic-tbp-mixin`)
relation
st

This method implements the specified functionality of its generic function.

`compute-st` [Generic function]
operation
relation
st
`&optional` *atomicp*

Given an activity *operation*, a relation (propagation reason) *relation*, and a start time

candidate *st*, this function will compute a new start time. The computation may cause reentrant calls (recursive invocations) to the propagator. If the optional parameter *atomicp* is true (it defaults to `nil`), the vertical propagator is not called to propagate the children of *operation* (i.e., *operation* is regarded an "atomic" activity).

`compute-st` [Method]
(*operation* `earliest-latest-mixin`)
relation
st
`&optional` *atomicp*

This is the default method. It ignores *atomicp* and will never call the vertical propagator.

`compute-st` [Method]
(*operation* `monotonic-tbp-mixin`)
relation
st
`&optional` *atomicp*

This method augments the functionality of the corresponding default method. If *operation* is `eq` to *relation* (which is the case when duration constraints are being propagated), it simply returns *st* (correctly assuming that the duration constraint was used when comupting *st* in the first place). It will also call the vertical propagator if *atomicp* is not true.

`find-st` [Generic function]
operation
connective
children

Given an activity *operation*, its connective tag *connective* (this is the value of the function `operation-connective`), and a list of its children (possibly empty), this function will find an appropriate start time for *operation*.

`find-st` [Method]
(*operation* `monotonic-tbp-mixin`)
connective
children

This method implements the specified functionality of its generic function.

`propagate-ft` [Generic function]
operation
relation
ft

This function is equivalent to `propagate-st` except that it operates on the end time

of *operation*.

`propagate-ft` [Method]
(*operation* `monotonic-tbp-mixin`)
relation
ft

This method implements the specified functionality of its generic function.

`compute-ft` [Generic function]
operation
relation
ft

This function is equivalent to `compute-st` except that it operates on the end time of *operation*. There is no option to disallow vertical propagation.

`compute-ft` [Method]
(*operation* `earliest-latest-mixin`)
relation
ft

This method implements the specified functionality of its generic function.

`compute-ft` [Method]
(*operation* `monotonic-tbp-mixin`)
relation
ft

This method augments the functionality of the default method (see the `compute-st` method for `monotonic-tbp-mixin` for an explanation). The default method will not call the vertical propagator, but this method will.

`compute-ft-reverse` [Generic function]
operation
st

This function may be called by `compute-ft` to reverse the direction of propagation, effectively allowing time bounds to be relaxed at the end of an activity graph/lineup. This function is called only if `tbp-reversible-p` returns true. The exact condition for this function to be called also requires the propagator to check for resource availability and the propagation direction to be backward.

This function will call `relax-latest-time-bounds` only if latest bounds are being propagated. In any case it will call the propagator and propagate forward using either the given start time *st* or the result of `relax-latest-time-bounds`.

`compute-ft-reverse` [Method]
(*operation* `monotonic-tbp-mixin`)
st

This method implements the specified functionality of its generic function.

`find-ft` [Generic function]
operation
connective
children

This function is equivalent to `find-st` except that it operates on the end time of *operation*.

`find-ft` [Method]
(*operation* `monotonic-tbp-mixin`)
connective
children

This method implements the specified functionality of its generic function.

`relax-latest-time-bounds` [Generic function]
operation

Relaxation happens in a situation where a time bound violation would occur otherwise. The purpose of this function is to relax the latest bounds of *operation* so they are not smaller that the earliest bounds. This function is called by `compute-ft-reverse`.

`relax-latest-time-bounds` [Method]
(*operation* `earliest-latest-mixin`)

This method implements the specified functionality of its generic function.

Bibliography

[1] James F. Allen, 1983. "Maintaining knowledge about temporal intervals", *Communications of the ACM*, 26:832–843.

[2] Marie Bienkowski, 1995. "Demonstrating the Operational Feasibility of New Technologies – the ARPI IFDs", *IEEE Expert*, February 1995, 27–33.

[3] Ora Lassila, 1995. "PORK Object System Programmers' Guide", Report CMU-RI-TR-95-12, Pittsburgh (PA), the Robotics Institute, Carnegie Mellon University.

[4] Ora Lassila, Marcel Becker and Stephen F. Smith, 1996. *An Exploratory Prototype for Reactive Management of Aeromedical Evacuation Plans*, Report CMU-RI-TR-96-03, Pittsburgh (PA), The Robotics Institute, Carnegie Mellon University.

[5] Ora Lassila, Markku Syrjänen and Seppo Törmä, 1991. "Coordinating Mutually Dependent Decisions in a Distributed Scheduler", in: Eero Eloranta (ed.), *Proceedings of the 4th IFIP TC5/WG5.7 International Conference on Advances in Production Management Systems - APMS'90*, Amsterdam (The Netherlands), Elsevier Science Publishers (North-Holland), 257–264.

[6] Claude LePape and Stephen F. Smith, 1987. "Management of Temporal Constraints for Factory Scheduling", *Proc. Working Conference on Temporal Aspects in Information Systems*, Paris (France), North-Holland.

[7] A.K.Mackworth, 1977. "Consistency in Networks of Relations", *Artificial Intelligence* 8, 99–118.

[8] Stephen F. Smith and Ora Lassila, 1994. "Configurable Systems for Reactive Production Management", in *Knowledge-Based Reactive Scheduling*, IFIP Transactions B-15, Amsterdam (The Netherlands), Elsevier Science Publishers.

[9] Stephen F. Smith and Ora Lassila, 1994. "Toward the Development of Mixed-Initiative Scheduling Systems", in: Mark H. Burstein (ed.), *ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative Workshop Proceedings*, San Francisco (CA), Morgan Kaufmann, 145–154.

[10] Guy L. Steele, Jr., 1990. *Common Lisp – the Language* (second edition), Bedford (MA), Digital Press.

[11] Jonathan Stillman, 1994. "Dual-use applications of Tachyon: From force structure modeling to manufacturing scheduling", in: *Proceedings of the 4th Annual 1994 IEEE Dual-Use Technologies & Applications Conference*, Institute of Electrical and Electronics Engineers, Utica (New York), May 1994, 114–123.

[12] Seppo Törmä, 1990. "Aikarajojen vyöryttäminen tuotannon ajoittamisessa" ("Propagation of Time Bounds in Production Scheduling", in Finnish), in Markus Djupsund et al (eds.), *STeP-90 – Finnish Artificial Intelligence Symposium*, Oulu (Finland), Finnish Artificial Intelligence Society.

[13] Seppo Törmä, Ora Lassila and Markku Syrjänen, 1991. "Adapting the Activity-Based Scheduling Method to Steel Rolling", in: G.Doumeingts, J.Browne and M.Tomljanovich (eds.), *Computer Applications in Production and Engineering: Integration Aspects (CAPE'91)*, Amsterdam (The Netherlands), Elsevier Science Publishers (North-Holland).

Index